Automated Synthesis of Adversarial Workloads for Network Functions

Luis Pedrosa
EPFL
luis.pedrosa@epfl.ch

Rishabh Iyer
EPFL
rishabh.iyer@epfl.ch

Arseniy Zaostrovnykh
EPFL
arseniy.zaostrovnykh@epfl.ch

Jonas Fietz
EPFL
jonas.fietz@epfl.ch

Katerina Argyraki
EPFL
katerina.argyraki@epfl.ch

ABSTRACT

Software network functions promise to simplify the deployment of network services and reduce network operation cost. However, they face the challenge of unpredictable performance. Given this performance variability, it is imperative that during deployment, network operators consider the performance of the NF not only for typical but also adversarial workloads. We contribute a tool that helps solve this challenge: it takes as input the LLVM code of a network function and outputs packet sequences that trigger slow execution paths. Under the covers, it combines directed symbolic execution with a sophisticated cache model to look for execution paths that incur many CPU cycles and involve adversarial memory-access patterns. We used our tool on 11 network functions that implement a variety of data structures and discovered workloads that can in some cases triple latency and cut throughput by 19% relative to typical testing workloads.

KEYWORDS

Network Function Performance; Adversarial Inputs

1 INTRODUCTION

This work is about software network functions (NFs): pieces of code, typically written in C or C++, that provide packet-processing functionality, such as forwarding, load balancing and network address translation. Traditionally, such functionality has been relegated to closed network appliances or middleboxes, often implemented in hardware. Recently, however, there has been a push towards software NFs, which have the potential to offer more flexibility, reduced time-to-market, and reduced capital and operating expenses [18, 34, 35].

This shift from hardware middleboxes to software NFs comes with the challenge of unpredictable performance. While hardware middleboxes process packets through ASICs that typically yield stable performance, software NFs process packets on general-purpose CPUs, which may yield variable performance. This variability provides an attack surface for adversaries seeking to degrade NF performance, e.g., by sending specially crafted packet sequences that significantly increase the per-packet latency and/or decrease throughput. Hence, when network operators deploy a new NF, they need to know its performance in the face of not only typical but also adversarial workloads; predicting NF performance assuming simple workloads, e.g., small packets with a uniform or Zipfian distribution of destination IP addresses [15], is useful but insufficient.

However, finding adversarial workloads in NFs—or any other non-trivial piece of software—can be hard. Different packet sequences can traverse different execution paths, with different performance envelopes. In some scenarios, finding the "bad paths" and the workloads that exercise them is relatively easy, e.g., when state is stored in a tree, in which case the adversarial workloads are those that update the tree in a way that induces skew. There are, however, more complicated scenarios, e.g., when state is stored in a hash table, in which case workloads that induce hash collisions can significantly degrade performance.

Our contribution is `CASTAN` (Cycle Approximating Symbolic Timing Analysis for Network Functions), a tool that automatically synthesizes adversarial workloads for NFs. Given the LLVM [2] code of an NF and a processor-specific cache model, `CASTAN` tries to discover execution paths that consume relatively large numbers of CPU cycles and synthesizes workloads that trigger them. We designed `CASTAN` with two properties in mind: (a) it should finish in useful time (minutes to hours); and (b) it should, ideally, discover workloads that are close to the worst-case scenario, even though we cannot formally guarantee that this will always be the case. The intended users of our tool are NF developers and network operators: developers can use `CASTAN`'s workloads

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230573

to stress-test the performance of their NFs and debug performance problems; operators can use them to provision their networks and be better prepared for worst-case scenarios.

CASTAN combines a symbolic execution engine with a processor-specific cache model. The main idea is to explore multiple execution paths of the NF while keeping an estimate of the execution cycles consumed by each path; then use directed symbolic execution [24] to guide exploration towards paths that are more likely to induce bad performance. The cache model helps find workloads that induce cache contention, resulting in frequent main-memory accesses. We have also developed a technique that helps reason about the performance of code constructs that are not amenable to analysis by symbolic execution.

To the best of our knowledge, this is the first work that leverages techniques from the programming languages and verification world to reason about NF performance and synthesize adversarial workloads. Prior work on NF code analysis focused on properties unrelated to performance: it identified semantic bugs [9, 22, 23, 29, 44], or formally proved crash-freedom and bounded execution [14], or memory safety and semantic properties [43]. Prior work on code performance analysis focused on code that uses a more constrained set of data structures [30, 32, 39].

We evaluated CASTAN with 11 NFs that employ a variety of data structures and algorithms. In scenarios where one can intuitively hand-craft adversarial workloads, CASTAN's workloads yielded very similar performance to the hand-crafted ones. In other, less intuitive scenarios, CASTAN's workloads increased latency by as much as a factor of 3 and decreased throughput by as much as 19% relative to typical testing workloads.

The rest of the paper is organized as follows: In §2 we provide background on symbolic execution. We describe CASTAN's design in §3, its implementation in §4, and its evaluation in §5. Finally, we discuss related work in §6 and conclude in §7.

2 BACKGROUND

In this section, we provide basic background on symbolic execution (*symbex*) [21], the technique that underlies CASTAN's analysis. The reader familiar with symbolic variables and expressions, symbex states, and path explosion, can safely skip to the next section.

Symbex is a program-analysis technique that explores multiple execution paths of a given program. It uses a special interpreter, called a symbolic execution engine (*SEE*). The SEE can make any input or variable (including a pointer) *symbolic*, i.e., assign to it a symbol representing many possible concrete values. As the symbolic inputs propagate through the program, the SEE keeps track of the resulting *symbolic expressions*. For example, suppose a program takes as input an integer *in*; the SEE can make *in* symbolic, assigning to it a symbol α that represents all possible integer values; if the program at some point assigns to an integer variable x the value $in + 1$, then x also becomes symbolic with value $\alpha + 1$. If the program reaches a conditional branch predicated on a symbolic value, the SEE can either concretize the symbolic variable/input, i.e., pick one of its possible concrete values, or explore all possible outcomes of the branch. For each outcome it explores, the SEE creates a new *execution state* and maintains a *path constraint*, which specifies the conditions that the symbolic inputs must satisfy such that the program reaches this execution state. For example, if an execution state has path constraint $in > 0$, this means that this execution state is reached if and only if the program's input *in* is positive. With the help of a solver, the SEE determines which path constraints are satisfiable and avoids exploring infeasible paths.

Symbolic execution suffers from path explosion: a very large, potentially unbounded number of paths to explore, which typically occurs in the presence of loops and/or code that maintains significant state. There are two general ways to side-step path explosion: constrain the input and/or the expressiveness of the code, or prioritize the exploration of certain execution states over others through "directed symbolic execution" [24], so as to achieve a specific goal, e.g., maximize line coverage [17, 33] or find specific kinds of bugs [29]. CASTAN falls in the latter category.

3 DESIGN

In this section, we describe CASTAN's design. We start with an overview (§3.1) and a description of our cache model (§3.2). Then we focus on the three main technical challenges we faced: identifying adversarial memory-access patterns (§3.3), identifying long execution paths (§3.4), and analyzing NFs with hash functions (§3.5).

3.1 Overview

We assume a reasonably powerful adversary: She has access to the NF code or intermediate build files, and she knows the processor on which the NF runs; this makes sense in the context of open-source NFs running in multi-tenant environments like cloud providers. We also assume that the adversary can generate adversarial workloads that can reach the targeted NF unmodified and unfiltered; this makes sense given the direct exposure of many NFs on the network.

CASTAN takes as input the LLVM code of an NF and a processor-specific cache model; the output is a sequence of N concrete packets, where N is a configurable parameter. Under the covers, we execute the given NF code on an SEE, providing as input a sequence of N symbolic packets. As symbex proceeds, we keep track of each execution state's "cost"—our expectation of the highest number of cycles per

packet (CPP) that this state can lead to—and prioritize exploring states with higher cost. When we exhaust our time budget, we halt the process, pick the state with the highest cost, provide its path constraint to a solver, and obtain a sequence of N concrete input packets that lead to this state; this packet sequence is our adversarial workload.

The main challenge is computing the cost of each state, which should reflect the likelihood of this state being a part of an adversarial workload. Consider a state $\mathcal{S}$ that results from the execution of a sequence $\mathcal{I}$ of LLVM instructions. We define $\mathcal{S}$'s cost as a sum of two parts: the "current cost" and the "potential cost." Ideally, the current cost would be the highest CPP that could be consumed *to reach* $\mathcal{S}$, while the potential cost would be the highest CPP that could be consumed *following* $\mathcal{S}$. To compute the former, we would consider all the memory access patterns that could result from executing $\mathcal{I}$ and pick the one that yields the highest CPP. Similarly, to compute the latter, we would identify all the instruction sequences that could follow $\mathcal{I}$ and all the memory access patterns that could result from executing each sequence, and pick the combination that yields the highest CPP. In practice, both the current and the potential cost are approximations, because we typically cannot consider all the feasible memory access patterns or instruction sequences.

Our approach is akin to an A* search [19], with the difference that we are trying to maximize, not minimize the expected cost. A traditional A* search finds the shortest path from a source to a destination, with the help of a heuristic that predicts the shortest distance from any candidate intermediate node to the destination. The result is guaranteed to be correct under the condition that the heuristic is "admissible," i.e., it returns a value that is always less than or equal to the actual shortest distance. To provide similar guarantees, we would need a heuristic that predicts the highest cost from any execution state to the point where the next packet is received and returns a value that is always *greater* than or equal to the actual highest cost. Given that we typically cannot tightly bound the highest cost (because we cannot tightly upper-bound the number of loop iterations in the code), our heuristic would often significantly over-estimate (in the extreme, return infinity), devolving into a breadth-first search and leading to path explosion. Hence, we preclude the formal guarantees of A*, in favor of finding useful adversarial workloads quickly.

Once we select the state with the highest cost, the next challenge is to resolve the associated path constraint and find a sequence of concrete packets that leads to this state. The default approach is to use an SMT solver, which we also do, but we need to overcome a hurdle: NF code often involves hash functions applied to packet headers, e.g., to compute a checksum, or to index a hash table that keeps per-flow state. As a result, resolving a path constraint may involve inverting one or more hash functions—something that an SMT solver typically fails to do in useful time. Sometimes we can solve the problem by reversing the hash function with precomputed rainbow tables [27] and reconciling the constraint on both the packet and the hash value. When that does not work, e.g., in case of a strong cryptographic hash function, we sidestep hash inversion altogether and output a sequence of partially symbolic packets (§3.5). In other words: if, to trigger bad NF performance, one has to invert a hash function, and if inverting that hash function is infeasible today, we still tell the developer what the bad performance is, and also provide a way to automatically synthesize a workload that will trigger it, should the hash function ever become invertible.

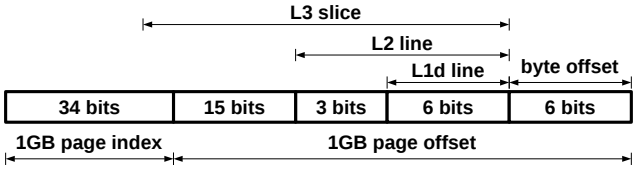

Figure 1: Memory hierarchy of Intel Xeon E5-2667v2.

3.2 Cache Contention Sets

To construct adversarial memory-access patterns, we need a model of the memory hierarchy. Unfortunately, building a detailed model is impossible, because the caching algorithms of modern processors are at least partially proprietary. In Xeon processors, in particular, the L1 and L2 cache locations are determined in the traditional way[1], however, the L3 cache slice is determined by a proprietary hash function. Fig. 1 illustrates what we know about the memory hierarchy of the processor used in our evaluation (Intel Xeon E5-2667v2), based on publicly available information and given that we used 1GB memory pages. In this processor, L1d is 8-way associative with 32kiB per core, L2 is 8-way associative with 256kiB, and L3 is 20-way associative with 25600kiB.

Hence, the only assumption we can make about our L3 cache is that a process's address space is divided into *contention sets*, i.e., sets of memory addresses such that: if an L3 cache with associativity α is empty, then α addresses from the same contention set can be brought into the L3 cache without any evictions, while bringing in an $(\alpha + 1)$st address from the same contention set will evict one of the α previously brought addresses.

Since the algorithm that determines the contention sets is proprietary, we developed a simple mechanism to reverse-engineer them. The main idea is to form different sets of memory addresses and measure each set's *probing time*, i.e.,

[1]The b_o least significant bits of a memory address are used to compute the cache-line offset, while the next b_1 and b_2 least significant bits, respectively, are used to index the L1 and L2 cache. The values of b_o, b_1 and b_2 can be inferred from the publicly disclosed cache sizes and associativity.

the time it takes to sequentially read all its addresses repeatedly in a loop (100 times in our case). To enforce sequential reads, i.e., avoid pipelining effects, we use pointer chasing [12]. Consider a set S_1 that includes at most α addresses from any contention set, and a set S_2 that includes at least $(\alpha + 1)$ addresses from some contention set. S_2's probing time will be higher than S_1's probing time by at least a *contention threshold* δ, as it includes an extra DRAM access.

Building on the above idea, we identify a contention set in three steps: (1) We form a set of addresses S that includes $\alpha+1$ addresses from a contention set C: Starting with an empty S, we keep adding addresses and measuring S's probing time, until we add some address A that increases S's probing time by more than the contention threshold δ. At this point, we know that A was the $(\alpha + 1)$st address added to S from the same contention set. We call that contention set C. (2) We reduce S such that it consists exactly of $\alpha + 1$ addresses from contention set C: For each address $A \in S$, we remove A from S and check whether that decreases S's probing time by more than δ; if yes, then we know that A belongs to C, and we add it back to S. Once we are done, all addresses in S belong to C. (3) We identify all remaining addresses that belong to the contention set C: For each memory address $A \notin S$, we replace an address in S with A and check whether that decreases S's probing time by more than δ; if not, then we know that A belongs to C. By repeating these steps multiple times, we identify all the contention sets of a given process.

In principle, different processes have different contention sets. This is because the L3 cache is physically indexed, i.e., the algorithm that determines the placement of a cache line in the L3 cache is applied to physical, not virtual addresses. In our Xeon processor, in particular, the L3 cache line is determined to some extent by bits 30–63 of the physical address (Fig. 1). We use 1GB memory pages, which means that bits 0–29 of each address are used as page byte offset, hence are the same between virtual and physical addresses. As such, some of the bits used to identify the L3 cache line are different between physical and virtual address, resulting in different contention sets per process.

To solve this problem, we repeat the contention-set discovery process with 8 different 1GB memory pages and across machine reboots. We post-process the results and retain only *consistent* contention sets, i.e., sets of addresses that have the same bits 0–29 and are always in the same contention set across different runs. This reduces the size of the discovered contention sets but produces results that generally hold across process runs and machine reboots. As different processors may use different proprietary hashing algorithms, the discovered contention sets may not hold across different processor types. For our Xeon processor, in particular, we discovered 23 409 contention sets varying in size from 32 to 5638 entries.

3.3 Current Cost and Memory Access

To compute the current cost of a state $\mathcal{S}$, corresponding to instruction sequence $\mathcal{I}$, we consider the sequence of all memory addresses accessed by $\mathcal{I}$ and try to constrain the symbolic memory addresses such that the resulting memory-access pattern incurs as many trips to main memory as possible. Then we consider each instruction i in $\mathcal{I}$ and assign to it an estimate of the number of cycles that it consumes: a fixed per-instruction cost learned empirically, if i does not access memory; and a fixed per-memory-level cost, if i does access memory. To constrain the symbolic memory addresses and determine which memory accesses are hits and misses, we use a cache model, initialized to a clear cache, that is built on top of the contention sets discovered as described in §3.2.

For example, consider an NF that accesses an IP lookup table, once per incoming packet. Which table location is accessed depends on the packet's IP headers, which, in our context, are symbolic. Hence, every table access yields an access to a new symbolic memory address A_s, constrained according to the boundaries of the table and the spacing between its entries. Moreover, at every table access, we use the cache model to determine: which (concrete) memory addresses are in the cache, which contention sets they belong to, and how many extra addresses from each contention set need to be accessed to cause an eviction. Based on this information, we create a list of candidate memory addresses that, if accessed, we expect to cause L3 cache contention. For each candidate memory address A, we use a solver to check whether A is compatible (satisfies the constraint associated) with the symbolic memory address A_s that is being accessed. If so, we concretize A_s to A and constrain the incoming packet's IP headers accordingly. Ideally, this constrains all symbolic memory addresses to the same contention set, which guarantees an L3 cache miss as soon as we exceed associativity. If that fails, we greedily try to constrain all addresses to as few contention sets as possible.

Limitations: To keep our approach scalable, we do not seek the provably worst memory-access pattern—just a very bad one that can be discovered within a reasonable time budget. (1) We do not consider the L1 and L2 caches; we tried designing a model of the memory hierarchy that did, but were unable to make it detailed and accurate enough to make a difference. (2) We do not consider prefetching and Data Direct I/O (DDIO) [1]. Prefetching is hard to model, because it is based on proprietary algorithms. However, as supported by our evaluation, prefetching does not significantly affect NF performance, because NF memory-access patterns are determined by network traffic and are typically not sequential or periodic. DDIO does affect NF performance, because it places the headers of incoming packets in the cache before they are accessed, thereby avoiding a previously mandatory

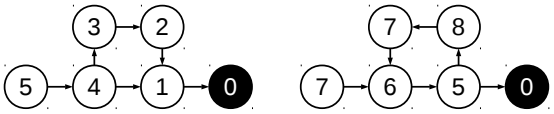

Figure 2: Example of annotated ICFGs showing each node's estimated maximum distance to the black node.

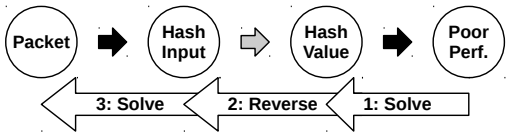

Figure 3: Handling hash functions: solid arrows show how NFs typically use hashing; empty arrows show CASTAN's reconciliation procedure.

cache miss. However, this improves all workloads the same, hence does not affect our task of seeking adversarial workloads. (3) When we constrain symbolic memory addresses, we make locally optimal decisions. Otherwise, we would have to follow the standard, non-scalable symbex behavior of resolving each accessed symbolic address to all feasible concrete addresses and creating a new state for each one, which fails to yield results in useful time.

3.4 Potential Cost

The potential cost of a state $\mathcal{S}$ is an estimate of the maximum number of cycles that could be consumed to get from $\mathcal{S}$ to the point where the next packet is received. To compute it, we rely on a pre-processing stage that extracts the NF's interprocedural control-flow graph (ICFG)[2] and annotates each node (which corresponds to an instruction) with an estimate of such a potential cost. This annotated ICFG then allows $\mathcal{S}$'s potential cost to be efficiently computed during symbex.

During pre-processing, we start from each node's local cost, assuming all memory accesses are L1 hits; we then use a special form of path-vector routing to propagate these local costs and estimate each node's potential cost[3]. In the absence of loops, this is simple: each node's cost is augmented by the cost of the most expensive successor. Fig. 2 on the left shows an example of an annotated ICFG, in the presence of a simple if-then-else statement.

Things are more complicated in the presence of loops: at this stage of the analysis we do not have enough context to bound the number of times a loop can execute; if we propagate costs naïvely, any loop will induce an infinite potential cost to every node within and before it, making the analysis intractable. To address this challenge, we ensure a node may show up at most M times in a path (within our path vector routing algorithm), where M is a configurable parameter. This essentially makes a static assumption that every loop executes exactly $M - 1$ times. In our evaluation, we use $M = 2$, which balances exploring the cost of a loop's internals ($M = 1$ hides all instructions within the loop body) against the negative effects of over-estimation. Fig. 2 on the right shows an example of an annotated ICFG, in the presence of a loop.

During symbolic execution, every time the SEE reaches a loop head, it creates two execution states: one that corresponds to exiting the loop as soon as possible, and one that corresponds to executing one more iteration (if that is feasible). Next, the SEE must choose which of the two states has the highest potential cost, and it always chooses the one that corresponds to executing one more loop iteration (again, as long as one more iteration is feasible). Hence, in the end, the SEE greedily explores the loop as deeply as possible.

Limitations: The ICFG cannot tell us which is the most expensive instruction sequence that follows a given state, because it does not take into account the constraints associated with that state. It only provides a first-order approximation. The time limit for the execution only allows for a partial state space exploration. Higher limits give a better chance of escaping local maxima, but remain an approximation when an exhaustive exploration is time-prohibitive. This prevents CASTAN from formally verifying worst-case performance.

3.5 Hash Functions

NFs that use data structures relying on hash functions pose a particular challenge to symbex. Symbexing a hash function typically leads to the creation of complex symbolic expressions that often exceed the solver's capabilities and result in solver timeouts.

We address this issue with a technique called *havocing* [5], which decouples the two parts of the code: the part that generates the input to the hash function, and the part that uses the hash value. This technique (which takes place at the gray arrow in Fig. 3) disables the execution of the hash function and replaces (havocs) its output with an unconstrained symbol, allowing the analysis of the code that uses the hash value to proceed normally, merely reasoning about constraints on the hash value. This results in a series of constraints that concern both the input packet and the resulting hash value. This on its own can already be useful, as it can provide insights into what a poor performance scenario may look like. The analysis essentially says that performance will suffer if we can find a packet that meets certain constraints and when hashed, produces a hash value that meets an additional set of constraints.

We reconcile the two sets of constraints in three steps

[2]The ICFG augments the traditional control-flow graph with function-call edges and typically takes less than a second to extract.

[3]A node's potential cost accounts for both calling functions in a chain (`a()` calls `b()`, and `b()` reaches the target) and returning from them (`a()` calls `b()`, which must return before `a()` reaches the target).

(shown as empty arrows in Fig. 3). We first use the solver to find a few candidate *hash values*. We then invert these hash values using brute-force methods augmented by the use of rainbow tables [27]. Finally, the solver checks if the hash input is compatible with the packet constraints and finds a matching packet.

While the first stage of this process is typically straightforward and likely to succeed, the likeliness of success of the second and third stages depends heavily on the quality of the rainbow tables. The process is analogous to solving constraints by successively attempting random assignments in the hope that a satisfying one eventually emerges. As we use rainbow tables, success first depends on the table having entries that match the hash-values found in the first stage. This requires the rainbow table to have enough entries that each value is represented a few times, i.e. $\sim 2^{\|hash-value\|}$. This doesn't pose a challenge as typical hash values are small ($\sim$ 20 bits, requiring a few millions of entries).

The more serious bottleneck is in finding hash inputs that satisfy the packet constraints. Finding satisfying values at random depends on how heavily constrained the packet is. For example, while analyzing a series of NFs that use a hash table (§5), we realized that the 8-bit IP protocol field was a part of the hash key. As the NFs only support TCP and UDP, the odds would reject $254/256 \approx 99\%$ of the rainbow table entries based on that constraint alone, potentially slowing the search and increasing the requisite table size by 100×. In such scenarios, we can increase the likelihood of success by generating a custom-tailored rainbow table with values that are more likely to fit the constraints. In this case, we populated the rainbow table with values that assume UDP.

4 IMPLEMENTATION

We implemented CASTAN [11] by forking an existing symbolic execution engine, KLEE [8], and adapting it to our needs. The key changes revolve around the implementation of the cache model (§3.3), the directed symbolic execution heuristic (§3.4), and the ability to havoc and reverse hash functions (§3.5). Additional tools were also created to help build, process, and validate the empirical cache models (§3.2) as well as to generate PCAP files from the analysis output.

The cache model is implemented as a special pluggable module which is called during the symbolic execution of the `load` and `store` memory operations. The module is designed as a plug-in so that multiple implementations can be easily swapped in. Our default cache model uses the contention sets discovered in §3.2. The module takes as input the symbolic expression of the pointer being used to access memory and adds constraints to the execution state. This processing operates in two stages. The initial phase looks at the current state of the cache model, picks the worst compatible cache line, and adds a series of constraints on the pointer expression to the path constraint, essentially concretizing the pointer. The second phase then takes this concrete pointer value and updates the cache model state so that future memory accesses will take it into account.

We implement our directed symbolic execution heuristic via a custom searcher class, which is a pluggable module that KLEE uses to pick which states to explore next. In this module, we first preprocess the NF LLVM code to extract the ICFG. Additionally, we annotate each instruction with cost estimates, as described in §3.4. Later, as the analysis is running, this annotated ICFG helps us to quickly compute the cost heuristic for each execution state, allowing us to order the pending states and prioritize the further exploration of those with a higher estimated CPP.

The ability to havoc and reverse hash functions is implemented in two phases. The first phase involves annotating the code to identify where the hash value is computed. The developer uses a special CASTAN annotation, `castan_havoc(input,output,expr)`. When the NF is compiled for use in production, this annotation simply equates to "`output = expr;`". When built for analysis, the annotation keeps track of the symbolic expression of `input` and then havocs the `output` variable by setting it to a new unconstrained symbol. Later, in a post-processing stage that occurs just before outputting the path, we use the information gathered through this annotation, alongside a user specified rainbow table, to reconcile the havocs, as explained in §3.5.

Finally, we modify KLEE to generate additional outputs that indicate the expected performance for each generated path. As such, a successful CASTAN run will generate two files for each path that it generates. The first is a traditional KTEST file, indicating concrete symbol values that will exercise the path. We convert this file into a PCAP file using a separate tool. The second file lists all of the CPU model metrics, on a per packet basis, including the number of non-memory instructions executed, the number of `loads` and `stores`, and the number of memory accesses that hit the cache. These metrics can be used directly to help debug the difference between distinct scenarios or simply to predict the performance envelope of each path, revealing the slowest path generated.

5 EVALUATION

In this section, we compare the workloads synthesized by CASTAN for 11 real NFs to workloads that were manually crafted to be adversarial by the engineer who wrote each NF. We first describe our evaluation setup (§5.1), then present our results for scenarios where the adversarial behavior comes primarily from memory accesses (§5.2), algorithmic complexity (§5.3), or hash-function manipulation (§5.4).

5.1 Setup

Testbed. Our testbed consists of a device under test (DUT) directly connected to a traffic generator/sink (TG). Both have Intel Xeon E5-2667v2 3.3GHz CPUs with 25.6MB of L3 cache and 32GB of RAM; they are connected over Intel 82599ES 10Gb NICs. In each experiment, the DUT runs one NF. The TG uses MoonGen [16] to replay specific PCAP files. Each experiment lasts 20 seconds. If a PCAP file does not yield 20 seconds worth of traffic, we replay it in a loop.

Performance metrics. First, we measure the end-to-end latency from the moment a packet exits until it re-enters the TG NIC. Our measurement process relies on hardware timestamps added by the TG NIC as described in [31], which has accuracy on par with hardware measurement devices. We do not ignore dropped packets: if the NF running on the DUT decides to drop a packet, we allow it to mark the packet as dropped, but we forward the packet back to the TG anyway, such that the latency it encounters is measured. During these experiments, the TG sends packets at a low enough rate that there is no more than one outstanding packet between the TG and the DUT, thus excluding any queuing or pipelining effects. We present the results in the form of a cumulative distribution function (CDF) per experiment.

Using an external TG benefits from high precision hardware timestamping, but this also measures the DPDK and driver stacks on the DUT as well as the transmission latency between the TG and the DUT. To quantify this overhead and estimate the NF latency in isolation we include in each plot the end-to-end latency CDF of a special NOP NF that forwards packets without any other processing. When we compare latency in relative terms, we use this NOP as a baseline to subtract from.

Second, we measure the maximum throughput achieved by each NF: we vary the rate at which the TG sends packets to the DUT and identify the highest rate at which the DUT drops less than 1% of the packets it receives.

Third, we conduct a micro-architectural characterization of each NF: we measure the number of reference cycles, instructions retired, and L3 cache misses (i.e., DRAM accesses) per packet, using CPU performance counter registers exposed through libPAPI [25]. These numbers allow us to reason about why one workload incurs worse performance than another. As in the latency experiments, there is no more than one outstanding packet between the TG and the DUT, and we present the results in the form of CDFs.

Network Function Logic. Our current research prototype assumes single-threaded NFs that use the basic DPDK API. This excludes many existing non-trivial open-source NFs. As such, we developed a library of NFs for evaluation purposes. We implemented three classes of NFs: IP longest prefix matching (LPM), source network address translation (NAT), and stateful L4 load balancing (LB).

LPM provides standard destination-based IP lookup. We populate the forwarding table with /8, /16, /24, and in some case /32 routes (depending on the underlying data structure), 8 of each. We chose the prefixes to overlap as much as possible, i.e., each prefix includes a more specific one (except for the /32 entries).

NAT provides standard source network address translation, i.e., it maintains per-flow state and uses it to: rewrite the source IP address and port number of packets coming from the internal network such that they appear to be coming from the NAT itself; rewrite the destination IP address and port number of packets coming from the external network such that they can be delivered transparently.

LB provides typical virtual IP (VIP) to direct IP (DIP) translation in a data-center network, i.e., it maintains per-flow state and uses it to: (1) rewrite the destination IP address of packets coming from the outside world such that they are transparently delivered to a backend server, ensuring all packets from the same connection go to the same server; and (2) rewrite the source IP address of packets coming from the data-center such that they appear to be coming from the LB itself. It picks backend servers for new connections in a round-robin fashion.

Data Structures. To test CASTAN's flexibility, for each NF class listed above, we use multiple implementations, each using a different underlying data structure, hence susceptible to different adversarial workloads.

We use three LPM implementations, each one striking a different balance between algorithmic complexity and memory efficiency: (1) The first one encodes the forwarding table in a Patricia trie [38], where each node of the trie corresponds to an IP prefix, and a node's children correspond to longer prefixes included in their parent. Lookup involves traversing the trie until we find the longest matching prefix. Hence, lookup complexity depends on the length of the longest supported prefix, which is 32 bits, in our case. (2) The second one implements Direct Lookup, where the forwarding table is translated into routes of equal-length IP prefixes (each as long as the longest supported prefix), which are then stored in a single, large array. In our case, the longest supported prefix consists of 27 bits, leading to an array that fits in a single 1GB page. Lookup involves indexing this array once. Relative to the first implementation, this one trades off memory efficiency for lower algorithmic complexity. (3) The third one is the LPM implementation that comes with DPDK, which implements a hierarchical version of Direct Lookup: the first 24 bits of the destination IP address are used to index a first-stage array, which then points to a second-stage array if any routes with longer IP prefixes exist within the given /24 IP prefix. Relative to the second implementation, this one reduces the memory footprint, while also limiting lookup

procedure to at most two array accesses.

The NAT and LB both store their per-flow state in an associative array. For each of these two NF classes, we use four different associative-array implementations: (1) A standard hash table of 65,536 entries that resolves collisions through separate chaining: elements that hash to the same position are stored in a linked list referenced by that position. Lookup of an element involves one hashing operation to pick a position, then traversing the referenced linked list until we find a matching element (or reach the end of the list). Hence, lookup complexity depends on the largest number of stored elements that happened to hash to the same position. (2) A standard hash ring, of 16.7M entries, that resolves collisions through open addressing: elements are stored in a circular array; if an element hashes to an array position that is already taken, it is stored in the first available subsequent position. The array is allocated within a single 1GB page, with each entry cache-aligned for performance. Lookup of an element involves one hashing operation to pick a position, then traversing the occupied positions of the array until we find a matching element (or traverse the entire array). Hence, lookup complexity depends on the number of stored elements. (3) An unbalanced binary tree, where lookup of an element involves a standard binary search. Without rebalancing, the tree is susceptible to skew, potentially becoming a linked list. Hence, lookup complexity depends on the number of stored elements. (4) The STL `std::map` data structure, which is a red-black tree, i.e., automatically re-balanced whenever skew occurs.

Workloads. First, we created generic workloads that we used across all NFs: (1) *1 Packet* consists of the same packet, replayed in a loop. We use it to assess best-case performance. (2) *Zipfian* consists of traffic that is randomly generated according to a Zipfian distribution with $s = 1.26$. The exponent s was computed from a public dataset [6] which includes real-world traffic captures from a University network. The corresponding PCAP file has 100,005 packets in 6,674 unique flows. It represents typical real-world traffic. (3) *UniRand* consists of traffic that is randomly generated according to a uniform distribution. The corresponding PCAP file has 1,000,472 packets in 1,000,001 unique flows. This kind of traffic is typically part of denial-of-service attacks and is used to stress-test NFs.

For the LB NFs, in particular, we did somewhat tailor the generic workloads to the NF in order to force the only interesting case, where the destination IP is set to the VIP. Any other traffic is either statically routed or outright dropped without any data structure access. This did not affect the resulting packet distribution parameters.

Second, we created NF-specific workloads: (1) CASTAN and (2) *Manual* are adversarial workloads generated, respectively, by CASTAN and by hand. (3) *UniRand* CASTAN is similar to *UniRand*, but involves the same number of flows as CASTAN. We use it for a fair comparison to CASTAN when sheer traffic volume is what matters for performance.

5.2 Adversarial Memory Access

First, we look at NFs that we expect to be susceptible to adversarial memory access and ask: can we craft workloads that consist of relatively few packets, yet introduce significant cache contention?

The NFs we consider are LPM with one-stage and two-stage Direct Lookup, which map the IP address space to a small number of large arrays. This approach restricts the number of instructions per packet to a small, predictable number, but introduces opportunity for cache contention. This is normally not a problem, as typical real-world workloads follow skewed, cache-friendly distributions, but could be a problem if an adversary can craft a workload that purposefully causes cache contention, especially if she does it with relatively few packets, i.e., without even filling the cache. The last point is important, because the smaller the workload an adversary needs to have an effect, the harder it is to detect it with standard entropy-based anomaly detectors.

For these two NFs, CASTAN synthesized workloads consisting of 40 unique packets, each in a different flow. We did not craft *Manual* workloads, as we were not able to reverse-engineer cache behavior by hand. The straightforward way to stress-test these NFs would be to access as much memory as possible, which *UniRand* already does.

Fig. 4 shows the latency CDF for LPM with single-stage Direct Lookup. This NF uses a single 1GB array, which far exceeds the size of the 25.6MB L3 cache. First, we see that the *Zipfian* workload experiences similar latency as *1 Packet*, indicating an insignificant cache-miss rate. Second, we see that the *UniRand* workload triples the latency introduced by the NF: the median distance between the NOP and *UniRand* curves is about three times the median distance between the NOP and *Zipfian*/*1 Packet* curves. This is consistent with the expectation that uniformly accessing the 1GB array will lead to a significant cache miss rate.

Most importantly, the 40-packet workload synthesized by CASTAN experiences similar latency as the 1M-packet *UniRand* workload. So, both CASTAN and *UniRand* triple latency, but CASTAN does it with four orders of magnitude fewer packets. *UniRand* CASTAN, which, in this case, is a *UniRand*-like workload that consists of 40 packets, introduces similar latency with *Zipfian* and 1 Packet.

Table 1 shows the throughput for the same NF. We see that CASTAN and *UniRand* achieve 19% lower throughput than the other workloads. The micro-architectural analysis confirms these results: Fig. 5 shows the CDF of the number of reference cycles consumed per packet, and clearly illustrates the difference between the typical *Zipfian* workload and the

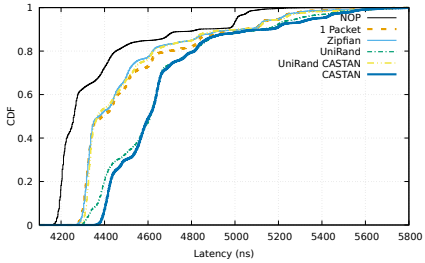

Figure 4: End-to-end latency CDF for LPM with 1-stage Direct Lookup.

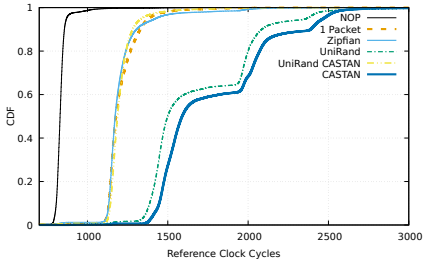

Figure 5: CPU reference cycles CDF for LPM with 1-stage Direct Lookup.

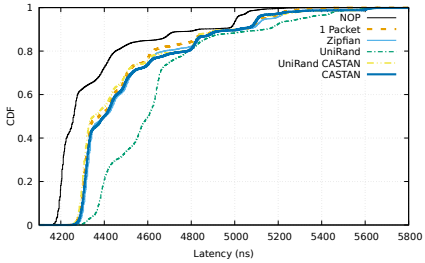

Figure 6: End-to-end latency CDF for LPM with 2-stage Direct Lookup.

NF	*LPM 1-stage DL*	*LPM 2-stage DL*	*LPM btrie*	*LB unbalanced tree*	*NAT unbalanced tree*	*LB red-black tree*	*NAT red-black tree*	*NAT hash table*	*LB hash table*	*NAT hash ring*	*LB hash ring*
NOP	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45	3.45
1 Packet	2.59	2.87	2.87	2.87	2.49	2.49	2.38	2.44	2.87	2.44	2.87
Zipfian	2.59	2.86	2.87	2.7	2.17	2.33	1.9	2.38	2.76	2.38	2.87
UniRand	2.12	2.49	2.8	1.64	0.95	1.32	0.95	0.47	1.48	1.96	2.65
UniRand CASTAN	2.59	2.87	2.87	2.65	2.28	2.6	2.28	2.33	2.87	2.44	2.87
CASTAN	2.1	2.82	2.65	2.69	2.01	2.56	2.22	2.39	2.73	1.97	2.69
Manual	-	-	2.7	2.7	1.9	-	-	-	-	-	-

Table 1: Maximum throughput measured for each NF under each workload (Mpps)

NF	*LPM 1-stage DL*	*LPM 2-stage DL*	*LPM btrie*	*LB unbalanced tree*	*NAT unbalanced tree*	*LB red-black tree*	*NAT red-black tree*	*NAT hash table*	*LB hash table*	*NAT hash ring*	*LB hash ring*
NOP	271	271	271	271	271	271	271	271	271	271	271
1 Packet	309	317	341	378	549	469	617	416	394	610	409
Zipfian	309	317	341	433	688	663	900	666	394	683	409
UniRand	309	317	341	1127	2271	1099	2054	1658	630	729	415
UniRand CASTAN	309	317	343	422	626	537	703	593	394	610	409
CASTAN	309	317	699	678	1100	559	769	593	468	610	409
Manual	-	-	699	678	1224	-	-	-	-	-	-

Table 2: Median instructions retired per packet for each NF under each workload.

NF	*LPM 1-stage DL*	*LPM 2-stage DL*	*LPM btrie*	*LB unbalanced tree*	*NAT unbalanced tree*	*LB red-black tree*	*NAT red-black tree*	*NAT hash table*	*LB hash table*	*NAT hash ring*	*LB hash ring*
NOP	1	1	1	1	1	1	1	1	1	1	1
1 Packet	2	2	2	2	2	2	2	1	2	2	2
Zipfian	2	2	2	2	2	2	2	2	2	2	2
UniRand	3	3	2	2	5	2	7	8	2	4	3
UniRand CASTAN	2	2	2	2	2	2	2	2	2	2	2
CASTAN	3	2	2	2	2	2	2	2	2	4	4
Manual	-	-	2	2	2	-	-	-	-	-	-

Table 3: Median L3 misses per packet incurred by each NF under each workload.

adversarial CASTAN workload. Moreover, the two workloads exhibit the same number of retired instructions per packet (Table 2), but different L3 cache misses per packet (Table 3), confirming that the CASTAN workload's worse performance is due to a higher cache miss rate.

The results are different for LPM with two-stage Direct Lookup. This NF uses a 64MB array in the first stage, which still exceeds the L3 cache, but not by orders of magnitude. Fig. 6 shows that all workloads except for *UniRand* experience similar latency. This is not surprising: On the one hand, CASTAN managed to find only 10 packets that could map to the same L3 cache location; this number is below cache associativity, which means that the CASTAN workload could not cause cache contention. On the other hand, the NF's data

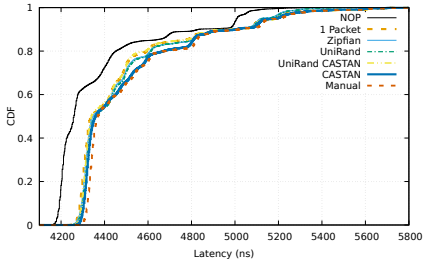

Figure 7: End-to-end latency CDF for LPM implemented with a Patricia trie.

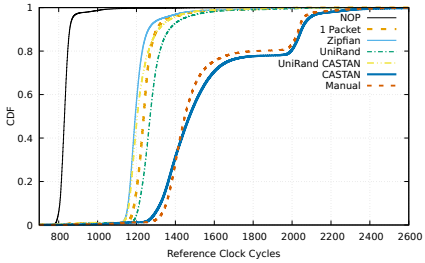

Figure 8: CPU reference cycles CDF for LPM implemented with a Patricia trie.

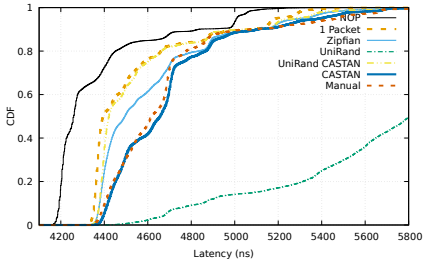

Figure 9: End-to-end latency CDF for NAT implemented with an unbalanced tree.

structures still exceed the L3 cache, which means that a large enough workload *could* cause cache contention—and indeed, *UniRand*, with its 1M packets and flows, does. Given enough time, we expect that CASTAN would also produce a workload large enough to cause cache contention, but it is not yet able to do it in useful time.

In conclusion, even though, in the common case, two-stage Direct Lookup is one-memory-access slower than one-stage Direct Lookup, it is more robust against performance attacks, because the smaller data structures make it harder to find small workloads that cause cache contention.

5.3 Algorithmic Complexity Attacks

Next, we look at NFs that we expect to be susceptible to adversarial workloads seeking to increase the number of instructions per packet.

We start from LPM with a Patricia trie. For this NF, CASTAN synthesized a workload of 30 packets and flows. We also crafted a *Manual* workload of 8 packets that match the most specific routes of the forwarding table, which results in traversing the longest paths of the trie. Upon inspection, we found that the CASTAN workload closely resembles the *Manual* one: in addition to finding packets that match the most specific routes, it also picked packets that are off by just one bit at the end, thus requiring the same amount of processing steps.

As we can see in Fig. 7, the 30-packet CASTAN workload experiences slightly worse latency than the 100K-packet *Zipfian*, and similar latency to the 1M-packet *UniRand* workload. Moreover, CASTAN experiences similar latency to *Manual* without the benefit from human insight. According to the micro-architectural analysis, CASTAN and *Manual* consume significantly more reference cycles (Fig. 8) and instructions (Table 2) per packet than the other workloads. This difference, however, did not translate into a significant difference in latency.

We also consider NAT and LB with unbalanced trees. For these NFs, CASTAN synthesized workloads consisting, respectively, of 50 and 30 packets and flows. We also crafted *Manual* workloads that skew the tree and turn it into a linked list; e.g., for NAT, such a workload consists of a sequence of packets with the same source and destination IP and source port, and increasing destination ports. Another way to stress-test these NFs is to increase the size of the tree as much as possible by sending a large number of flows, which is what the *UniRand* workload does.

Fig. 9 shows the latency CDF for NAT (the results for LB are similar). The most interesting result is that the 50-packet CASTAN workload experiences 67% worse median latency than the 100K-packet *Zipfian*, though it cannot beat the 1M-packet *UniRand*. *UniRand* experiences more latency than *Zipfian*, simply because it has an order of magnitude more flows, hence creates a larger tree. CASTAN, on the other hand, experiences more latency than *Zipfian* with two orders of magnitude *fewer* flows, because it creates an unbalanced tree. Moreover, CASTAN experiences similar latency to *Manual* without the benefit of human insight.

The micro-architectural analysis confirms these results: The number of reference cycles per packet approximately mirrors latency (Fig. 10). Moreover, all workloads experience similar L3 cache misses per packet (Table 3), but different numbers of retired instructions per packet (Table 2), confirming that, for this NF, performance differences are due mostly to algorithmic complexity, not cache contention.

Not surprisingly, if we replace the unbalanced tree with a Red-Black tree, CASTAN fails to find an adversarial workload, and latency experienced depends simply on the total number of flows (which determine the size of the tree). This is illustrated in Fig. 11, which shows the latency CDF for NAT with a Red-Black tree (the results for LB are similar): The 1M-flow *UniRand* experiences worse latency than the 6K-flow *Zipfian*, which experiences worse latency than the 50-flow CASTAN. Internally, this kind of code induces local maxima within the CASTAN analysis. As the analysis selects states that make the tree deeper, the rebalancing algorithm kicks in and thwarts the attempt. In the end, CASTAN explores many states with mostly similar costs and picks the worst among the almost equal candidates.

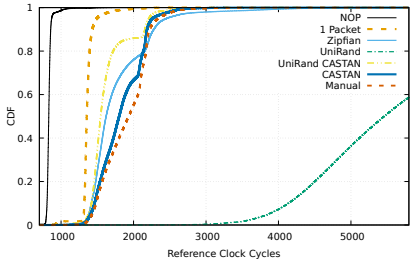

Figure 10: CPU reference cycles CDF for NAT implemented with an unbalanced tree.

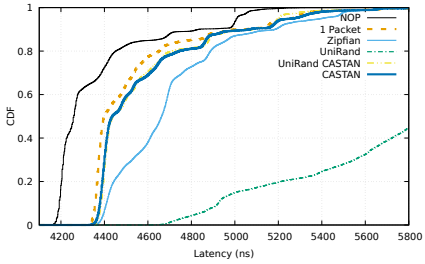

Figure 11: End-to-end latency CDF for NAT implemented with a red-black tree.

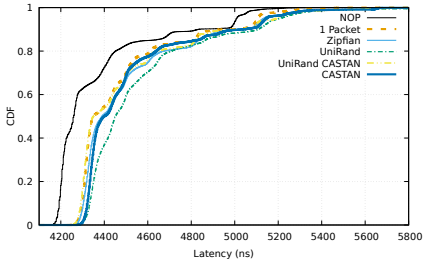

Figure 12: End-to-end latency CDF for LB implemented with a hash table.

5.4 Cracking Hash Functions

Finally, we look at NFs that use hash functions for indexing their data structures and ask: can we craft workloads that consist of relatively few packets, yet introduce a significant rate of hash collisions?

We start from two LB NFs, one with a 65,536-entry hash table, and one with a 17M-entry hash ring. For the former, CASTAN synthesized a workload of 30 packets that cause persistent hash collisions. This workload experiences slightly worse latency than the 100K-packet *Zipfian*, and slightly better latency than the 1M-packet *UniRand* (Fig. 12). In the case of the hash ring, however, the dominant adversarial behavior caused by the CASTAN workload is cache contention. This is because the sheer size of the hash ring makes it vulnerable to adversarial memory access, and CASTAN found it easier to synthesize packets that contend for the same L3 cache location than packets that cause collisions. The CASTAN workload experiences 56% worse median latency than *Zipfian* and 32% worse median latency than *UniRand* (Fig. 13).

We also consider NAT NFs that use the same data structures. These pose a particularly difficult challenge to the CASTAN analysis, as the NAT hashes and stores two entries for each flow, using different parts of the packet to form different keys (one to match outgoing packets and another to match returning traffic). The challenge lies in the fact that while both hashes are independently havoced, the keys that serve as inputs are related and share some portion (the external end-point's IP and port). This means that CASTAN not only has to find an entry in the rainbow table that reverses the hash, but actually two of them that reverse two different hashes while preserving the relationship between keys and also satisfying the constraints on flow uniqueness. In practice, CASTAN was rarely able to do this reliably. For the hash table, the complex set of constraints for the related keys and packet uniqueness while also trying to cause systematic collisions defeated CASTAN, as none of the havocs were successfully reversed. For the hash ring, the fact that each havoc reverses a unique value made the problem somewhat easier. As a result, we were able to systematically reverse the first of the two havocs used for each packet, while satisfying all uniqueness constraints. The second one remained unreconciled, as we could not find entries in the rainbow table that both reversed the second hash and had a key that was related to the first one in the expected manner.

NF	*# Packets*	*Time (seconds)*
LB / Hash table	30	115
LB / Hash ring	40	31955
LB / Red-Black Tree	30	437
LB / Unbalanced Tree	30	453
LPM / Patricia Trie	30	1166
LPM / Lookup Table	40	2542
LPM / DPDK LPM	40	88508
NAT / Hash Table	30	5210
NAT / Hash ring	40	2040
NAT / Red-Black Tree	35	6836
NAT / Unbalanced Tree	50	2444

Table 4: List of NFs, indicating how many packets we generated and the analysis run time.

The results reflect this outcome: For the NAT with hash table, the CASTAN workload experiences slightly worse latency than the *Zipfian*, but significantly better latency than the *UniRand* (Fig. 14). For the NAT with hash ring, the fact that one of the two hashes is successfully reversed for each packet allows part of the expected slowdown to be achieved (the one that results from the first access to the data structure). As a result, the CASTAN workload experiences 159% worse median latency than *Zipfian* and 89% worse median latency than *UniRand* (Fig. 15).

5.5 Discussion

Through our measurement campaign, we were able to show that CASTAN is quite capable of generating useful adversarial workloads. In Table 5, we summarize the key results, showing how each NF is affected by typical and adversarial workloads, including a manually crafted one and the one generated by CASTAN. Table 4 shows how long it took for CASTAN to generate these workloads. The results show that when it was possible to manually create an adversarial workload using human intuition, CASTAN closely matched it's performance,

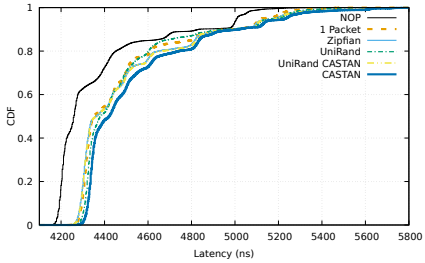

Figure 13: End-to-end latency CDF for LB implemented with a hash ring.

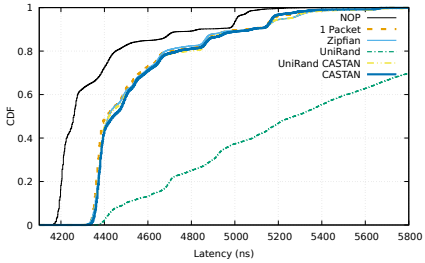

Figure 14: End-to-end latency CDF for NAT implemented with a hash table.

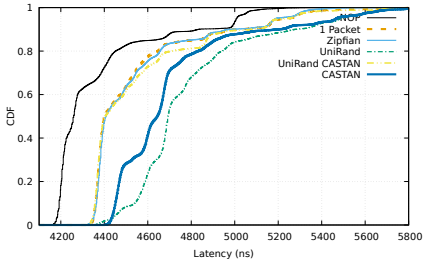

Figure 15: End-to-end latency CDF for NAT implemented with a hash ring.

NF	*Median Deviation (ns)*		
	Zipfian	*Manual*	CASTAN
LB / Hash table	131	-	141
LB / Hash ring	103	-	161
LB / Red-Black Tree	179	-	141
LB / Unbalanced Tree	109	256	240
LPM / Patricia Trie	87	112	100
LPM / Lookup Table	115	-	346
LPM / DPDK LPM	141	-	141
NAT / Hash Table	160	-	182
NAT / Hash ring	148	-	384
NAT / Red-Black Tree	404	-	176
NAT / Unbalanced Tree	237	359	397

Table 5: List of NFs and tested workloads, indicating the median latency deviation from *NOP*.

automatically. When the structure of the NF makes it more difficult to manually reason in such a way, CASTAN proves to be invaluable, generating workloads up to 201% slower than typical *Zipfian* traffic.

Due to limitations in our experimental setup, our evaluation only explored scenarios with 100% adversarial traffic. A more realistic adversary can only inject a fraction of the overall traffic as a part of a DDoS campaign. We expect that due to the effects of head of line blocking, even a limited adversary could potentially cause more damage than their limited capabilities would suggest. Furthermore, whereas a typical DDoS overwhelms the NF through sheer volume, adversarial workloads can increase the efficiency of such an attack by consuming disproportionately more resources for the same amount of attack traffic. Studying such effects would require a detailed cost-benefit analysis from the attacker's point of view, which we leave to future work.

CASTAN also has several limitations. For simpler network functions with a more direct mapping between the input packet and its processing performance, CASTAN has an easier time reverse engineering adversarial workloads. However, there are several ways in which this performance envelope can be obfuscated, making it more difficult for us to derive such workloads. The use of one-way functions, such as hash functions poses one such challenge. We use rainbow tables to help reverse these but this can fail or only partially succeed when additional constraints on the hashed packet come into play. For now, analyzing the hash function directly in symbolic execution is intractable but these functions are not typically cryptographically secure and can hence be reasoned about with sufficiently powerful solvers or appropriate constraint algebra. We leave exploring such possibilities to future work.

Another challenge arises when the constraints on symbolic pointers that arise during the CASTAN analysis prove to be incompatible with the limited contention sets that our cache model was able to capture empirically. To handle this more robustly, it would be more appropriate to have a more complete model of the cache behavior, based on the actual cache slicing and eviction algorithms implemented in the CPU. Reverse engineering the internal structure of CPU caches to this effect is still in many cases an open problem in active research [4]. We tried incorporating some prior art from this field in our own models but failed to achieve sufficient predictive power for our purposes, hence our reliance on empirical models. If it ever becomes possible to further encapsulate the cache behavior in a more powerful algebraic model that can be integrated into CASTAN, we anticipate that it will be much easier to generate adversarial workloads with even more precision (i.e. even fewer packets).

6 RELATED WORK

Performance evaluation and diagnosis:

Software performance attacks are a well studied problem. [13] describes adversarial complexity-based attacks on data-structures and network applications and how to mitigate them. [3, 36] have studied specific IDS NFs while considering both algorithmic complexity and the cache. These works manually study specific systems; whereas CASTAN offers an automated approach to discovering such issues.

[30, 32, 39] use fuzzing-like approaches to automatically expose performance bottlenecks at the level of individual methods and data-structures. Such approaches may not scale well when the input space is larger and less structured as is the case in NFs. [28] automatically detects and exploits

second order denial-of-service attacks in web services. Such attacks result from the implicit complexity of processing database queries in web service architectures and don't directly apply in NF environments. Like CASTAN, [7] also uses symbex to attack algorithmic complexity. This work uses exhaustive symbex for small inputs to find worst-case inputs and then generalizes to discover likely worst-case candidates for larger inputs. CASTAN could benefit from this technique for more complex NFs where the analysis may otherwise not scale.

Quantifying worst-case performance can follow one of two approaches. The first approach, of which CASTAN is an example, under-approximates the worst case performance. By constructing adversarial workloads this approach provides a lower bound on the worst-case performance. The other approach, commonly known as Worst-Case Execution Time (WCET) Analysis [40] typically over-approximates the WCET and provides conservative but safe upper bounds on the execution time. While such an approach may provide formal performance guarantees, it does not provide an adversarial workload which is of prime importance during the debugging phase.

With the adoption of software NFs, there exist several proposals for online performance diagnosis systems. *NFVPerf* [26] leverages passive traffic monitoring to identify both hardware and software bottlenecks in Virtualized NFs. *PerfSight* [41] leverages low-level packet processing performance metrics to detect and diagnose performance problems. These systems help diagnose performance issues at run time given a specific NF workload. CASTAN complements these approaches by generating adversarial workloads that they can then use to diagnose and debug the underlying performance issue.

Program Analysis Applied to NFs: Several prior works have proposed using static analysis to help understand, debug, and verify software NFs. StateAlyzr [20] does this to identify per-flow and global state in NFs to facilitate the implementation of state migration and redistribution. Many other approaches, like CASTAN, use symbolic execution to find bugs or formally verify correctness. [9, 10, 22, 44] leverage this technique to automate bug finding and test-case generation. [14, 43], on the other hand, use exhaustive symbolic execution to formally verify functional correctness. Others have extended symbolic execution to explore multiple systems at once: [23] finds discrepancies between different SDN agents, while [29] identifies interoperability issues between a client and a server. [37] symbolically executes NF models to reason about network properties like reachability and loops. Right now there is no guarantee that NF models and implementations agree, but emerging techniques automatically synthesize the models [42].

7 CONCLUSIONS

In this paper, we present CASTAN, a tool that automates the generation of adversarial workloads for NFs that lead to poor performance. We statically analyze the NF code using symbolic execution to find code paths that perform poorly. During analysis, we attack NF performance on three fronts: algorithmic complexity, adversarial memory access patterns, and reversing hash functions. Algorithmic complexity attacks are caused by code paths which execute more instructions. We find these paths by using a directed symbolic execution heuristic which estimates the number of CPU cycles needed to process each packet. We then prioritize the exploration of code paths that maximize this metric. We also build a CPU cache model that allows us to determine specific memory access patterns that induce persistent L3 cache misses and evictions. Finally, we incorporate the use of rainbow tables during analysis to reverse one-way functions such as those used in hash tables. These three techniques combined allow CASTAN to successfully generate adversarial workloads for 11 different NFs that we evaluate in a detailed measurement campaign in §5. The results show that under ideal circumstances, a CASTAN workload is able to increase NF latency by 201% and decrease throughput by 19% when compared to typical test network traffic. When the NF structure is simple enough that human intuition can create adversarial workloads manually, we show that a corresponding CASTAN workload behaves similarly, while being generated in an automated fashion. We also show that CASTAN completes in a reasonable amount of time, typically less than an hour.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Sujata Banerjee for helping us improve our work. We were funded by Starting Grant #BSSGI0_155834 from the Swiss National Science Foundation and Intel Corporation.

REFERENCES

[1] 2012. Intel Data Direct I/O Technology Overview. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf. Accessed: 2018-06-25.

[2] 2018. The LLVM Compiler Infrastructure. https://llvm.org/. Accessed: 2018-06-14.

[3] Yehuda Afek, Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. 2016. Making DPI Engines Resilient to Algorithmic Complexity Attacks. *IEEE/ACM Trans. on Networking* 24, 6 (2016).

[4] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. *IACR Cryptology ePrint Archive* 2015 (2015).

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*.

[6] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conf.*

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *Intl. Conf. on Software Engineering*.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symp. on Operating Sys. Design and Implem.*

[9] Marco Canini, Dejan Kostic, Jennifer Rexford, and Daniele Venzano. 2011. Automating the testing of OpenFlow applications. *Intl. Workshop on Rigorous Protocol Engineering* (2011).

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Symp. on Networked Systems Design and Implem.*

[11] CASTAN 2018. `CASTAN` code repository. https://github.com/nal-epfl/castan.

[12] Sophie Cluet and Claude Delobel. 1992. A general framework for the optimization of object-oriented queries. *ACM SIGMOD Record* 21, 2 (1992).

[13] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symp.*

[14] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *Symp. on Networked Systems Design and Implem.*

[15] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. 2012. Toward Predictable Performance in Software Packet-Processing Platforms. In *Symp. on Networked Systems Design and Implem.*

[16] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conf.* https://doi.org/10.1145/2815675.2815692

[17] Patrice Godefroid. 2012. Test Generation Using Symbolic Execution. In *IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science*, Vol. 18. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.24

[18] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (Feb 2015). https://doi.org/10.1109/MCOM.2015.7045396

[19] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Systems Science and Cybernetics* 4, 2 (July 1968). https://doi.org/10.1109/TSSC.1968.300136

[20] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Symp. on Networked Systems Design and Implem.*

[21] J. C. King. 1976. Symbolic Execution and Program Testing. *J. ACM* 19, 7 (1976).

[22] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. 2011. Finding protocol manipulation attacks. *SIGCOMM Computer Communication Review* 41, 4 (2011).

[23] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Intl. Conf. on Emerging Networking Experiments and Technologies*.

[24] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In *Intl. Static Analysis Symp.*

[25] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[26] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. 2016. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *IEEE Conf. on Network Function Virtualization and Software Defined Networks*. https://doi.org/10.1109/NFV-SDN.2016.7919491

[27] Philippe Oechslin. 2003. Making a faster cryptanalytic time-memory trade-off. In *Annual Intl. Cryptology Conf.*

[28] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. In *Conf. on Computer and Communication Security*. https://doi.org/10.1145/2810103.2813680

[29] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. Analyzing Protocol Implementations for Interoperability . https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa. In *Symp. on Networked Systems Design and Implem.*

[30] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security*.

[31] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2017. How to Measure the Killer Microsecond. In *ACM SIGCOMM Workshop on Kernel-Bypass Networks*.

[32] P. Puschner and R. Nossal. 1998. Testing the Results of Static Worst-Case Execution-Time Analysis. In *Real-Time Systems Symp.*

[33] C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. 1976. On the Automated Generation of Program Test Data. *IEEE Trans. on Software Engineering* 2, 4 (1976).

[34] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Symp. on Networked Systems Design and Implem.*

[35] Vyas Sekar and Petros Maniatis. 2011. Verifiable Resource Accounting for Cloud Computing Services. In *Cloud Computing Security Workshop*. https://doi.org/10.1145/2046660.2046666

[36] Randy Smith, Cristian Estan, and Somesh Jha. 2006. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conf.*

[37] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.*

[38] Wojciech Szpankowski. 1990. Patricia Tries Again Revisited. *J. ACM* 37, 4 (Oct. 1990). https://doi.org/10.1145/96559.214080

[39] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Intl. Symp. on Code Generation and Optimization*. https://doi.org/10.1145/3168830

[40] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008). https://doi.org/10.1145/1347375.1347389

[41] Wenfei Wu, Keqiang He, and Aditya Akella. 2015. PerfSight: Performance Diagnosis for Software Dataplanes. In *Internet Measurement Conf.* https://doi.org/10.1145/2815675.2815698

[42] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic Synthesis of NF Models by Program Analysis. In *ACM Workshop on Hot Topics in Networks*. https://doi.org/10.1145/3005745.3005754

[43] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *ACM SIGCOMM Conf.*

[44] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic test packet generation. In *Intl. Conf. on Emerging Networking Experiments and Technologies*.

A REPRODUCIBILITY

The CASTAN source code, including the analysis tool and the NFs used during the evaluations (§5), is publicly available on GitHub at: https://github.com/nal-epfl/castan

Reproducing the results in this paper involves two stages: i) analyzing the NFs to generate PCAP files with adversarial workloads, and ii) measuring NF performance under a variety of workloads, including the generated CASTAN workloads, to compare outcomes. README.md describes how to build and use CASTAN, including all of the steps necessary to generate the adversarial PCAP files and measure performance on a test-bed.